Yevgeniya Okuneva
STAT 242
HW5

Parallel Computing and Airline Delays Data

Parallel Computing and Airline Data

Introduction

In this exercise, we explore the parallel package to speedup the computations by running them in parallel. Here we focus on parallelizing the computations locally on the cores present on the master. Parallelizing computations using different machines with ssh will be applied to simulations in the final project.

Random Sample

We take a random sample of observations from each year of the data with n = 100,000. The idea is to parallelize the computations to gain speed. However, in order to get a better idea of how much speed we are actually gaining with parallelizing, we first obtain our random sample by implementing supposedly computationally efficient function where we create a list with 22 names of files we are going to sample from. Next, we loop through the list of files and obtain a random sample of 100,000 without replacement from each file storing the results in a single data frame. Table 1 provides system time that it took for the computations to complete. Pretty long. At the same time, considering the fact that we sampled 100,000 rows from each of the 22 files which gave us a sample of size 2,200,000, it wasn't terribly long.

Table 1. System time for obtaining random sample with lapply

```
    user   system  elapsed
1142.316    9.629 1173.547
```

We next implement parallel package to obtain sample by simply changing lapply() to mclapply() as suggested in “Parallel R” book. It turns out that the time required to complete the job almost doubled which is an indication of a need for different method or some modification to the function. One of the reasons to why the function took longer could be the fact that some files are larger some are smaller which means that some workers are done earlier and they simply sit and wait for more job. Next we try parLapplyLB(). This function introduces the load balancing where first n jobs are sent on the n workers, when the worker is done processing the job, the next job is sent to this worker not waiting until other workers complete their jobs. Unfortunately, sampling didn't work well with this approach.

Note: in this and subsequent approaches we use RNGkind random number generator function in R and set seed so clusters will have different sets of random numbers but so that the results are reproducable.

Next, we try a different approach were we first read data into R and then use parSapply to sample each year. No success with this approach either as it takes more time to load data into R then obtaining random sample for each year with the very first approach.

With a bit more thinking, no wonder these functions take more time as they work line by line. We turn to clusterApply and clusterApplyLB functions but both also take an enormous amount of time.

Failure of the above approaches could happen due to the way data is sent on the nodes. It is possible that all 22 data files are sent on each of the workers first which creates dramatic overload.

It was also attempted to obtain a random sample through Sqlite database however, the main limitation of databases is that it is a set-oriented language and it wants to do all the operations at once and there is no randomize operator which means we can't directly pick random rows. We could use pseudo-number generator function but the limitation is that it may be different for each sql extension.

Another approach to speed up the sampling would be to use shell commands. Introducing stratified sampling is also possible when using shell commands mixed with R commands. First, write shell command that counts the number of lines in the csv file with wc -l command. Then we would sample 100000 numbers in the range of the number of lines using R command. This would be saved and then this line numbers will be pulled from the csv files. Then we use pipe() connection to bring the data into R.

Random Forest in parallel

Due to the limitation of random forest function where it can't use the variables with more then 32 factors, we decided to shrink down the number of variables to day of the week, departure delay time and distance with response variable ArrDelay time.

In order to generate random numbers and obtain reproducible results we use the same approach as in previous section setting the seed with RNGkind and clusterSetRNGSreatm commands.

In order to minimize data transfer from master to the workers, we first load the data in the workspace. Next we make cluster on the local machine. Each cluster inherits the local workspace which can be seen with ClusterEvalQ(cl, ls()) function. We use the same data on each cluster, so in order not to transfer data with each new computation job, we use ClusterCall (vs. clusterApply where data is sent every time).

Even though the above approach was implemented to make computations in parallel and supposedly speed up the process, there is apparently some big issue that makes these computations slower then if they are obtained using a single CPU, the master.

Please note that we never obtained the results from the last two functions in the code (for randomForest) as every time we attempted to run them, the computer would freeze after few minutes, the workers CPU would get 102% overloaded and the only way to get the machine out of that is to manually shut it down.

The exploration of parallel computing will be continued in the final project where simulations for power analysis for 3 level mixed effects models will be ran in parallel on different machines.

GitHut Repository

Can be found at https://github.com/yokuneva/HW4.stat242